# App:

## Open()

open(devname, O\_RDWR | O\_NONBLOCK);

static int uvc\_v4l2\_open(struct file \*file)

{

struct video\_device \*vdev = video\_devdata(file);

struct uvc\_device \*uvc = video\_get\_drvdata(vdev);

struct uvc\_file\_handle \*handle = kzalloc(sizeof(\*handle), GFP\_KERNEL);

v4l2\_fh\_init(&handle->vfh, vdev);

v4l2\_fh\_add(&handle->vfh);

handle->device = &uvc->video;

file->private\_data = &handle->vfh;

**uvc\_function\_connect**(uvc);

}

void uvc\_function\_connect(struct uvc\_device \*uvc)

{

ret = **usb\_function\_activate**(&uvc->func);

}

int usb\_function\_activate(struct usb\_function \*function)

{

struct usb\_composite\_dev \*cdev = function->config->cdev;

**//cdev->deactivations = 1, function->name = uvc**

if (WARN\_ON(cdev->deactivations == 0)) status = -EINVAL;

else {

cdev->deactivations--;

if (cdev->deactivations == 0)

status = **usb\_gadget\_connect**(cdev->gadget); //gadget->ops->pullup(gadget, 1); // **dwc3\_gadget\_pullup()**

}

}

## dwc3\_thread\_interrupt()

static irqreturn\_t dwc3\_thread\_interrupt(int irq, void \*\_dwc)

{

struct dwc3 \*dwc = \_dwc;

for (i = 0; i < dwc->num\_event\_buffers; i++) {

struct dwc3\_event\_buffer \* evt = dwc->ev\_buffs[i];

left = evt->count;

if (!(evt->flags & DWC3\_EVENT\_PENDING)) continue;

while (left > 0) {

union dwc3\_event event;

event.raw = \*(u32 \*) (evt->buf + evt->lpos);

**dwc3\_process\_event\_entry(dwc, &event);**

evt->lpos = (evt->lpos + 4) % DWC3\_EVENT\_BUFFERS\_SIZE;

left -= 4;

dwc3\_writel(dwc->regs, DWC3\_GEVNTCOUNT(i), 4);

}

evt->count = 0;

evt->flags &= ~DWC3\_EVENT\_PENDING;

ret = IRQ\_HANDLED;

}

}

## dwc3\_process\_event\_entry()

static void dwc3\_process\_event\_entry(struct dwc3 \*dwc,const union dwc3\_event \*event)

{

if (event->type.is\_devspec == 0) return **dwc3\_endpoint\_interrupt**(dwc, &event->depevt);

switch (event->type.type) {

case DWC3\_EVENT\_TYPE\_DEV: dwc3\_gadget\_interrupt(dwc, &event->devt); break;

default: dev\_err(dwc->dev, "UNKNOWN IRQ type %d\n", event->raw);

}

}

## dwc3\_endpoint\_interrupt()

static void dwc3\_endpoint\_interrupt(struct dwc3 \*dwc, const struct dwc3\_event\_depevt \*event)

{

struct dwc3\_ep \*dep;

u8 epnum = event->endpoint\_number;

dep = dwc->eps[epnum];

if (epnum == 0 || epnum == 1) { **dwc3\_ep0\_interrupt**(dwc, event); return; }

switch (event->endpoint\_event) {

case DWC3\_DEPEVT\_XFERCOMPLETE:

dep->resource\_index = 0;

if (usb\_endpoint\_xfer\_isoc(dep->endpoint.desc)) { return; }

dwc3\_endpoint\_transfer\_complete(dwc, dep, event, 1);

break;

case DWC3\_DEPEVT\_XFERNOTREADY:

if (usb\_endpoint\_xfer\_isoc(dep->endpoint.desc)) { dwc3\_gadget\_start\_isoc(dwc, dep, event); }

else { ret = \_\_dwc3\_gadget\_kick\_transfer(dep, 0, 1); }

break;

}

}

## dwc3\_ep0\_interrupt()

void dwc3\_ep0\_interrupt(struct dwc3 \*dwc, const struct dwc3\_event\_depevt \*event)

{

u8 epnum = event->endpoint\_number;

//Transfer Complete while ep0out in state 'Setup Phase'

switch (event->endpoint\_event) {

case DWC3\_DEPEVT\_XFERCOMPLETE: **dwc3\_ep0\_xfer\_complete**(dwc, event); break;

case DWC3\_DEPEVT\_XFERNOTREADY: dwc3\_ep0\_xfernotready(dwc, event); break;

case DWC3\_DEPEVT\_XFERINPROGRESS:

case DWC3\_DEPEVT\_RXTXFIFOEVT:

case DWC3\_DEPEVT\_STREAMEVT:

case DWC3\_DEPEVT\_EPCMDCMPLT:

break;

}

}

## dwc3\_ep0\_xfer\_complete()

static void dwc3\_ep0\_xfer\_complete(struct dwc3 \*dwc, const struct dwc3\_event\_depevt \*event)

{

struct dwc3\_ep \*dep = dwc->eps[event->endpoint\_number];

dep->flags &= ~DWC3\_EP\_BUSY;

dep->resource\_index = 0;

dwc->setup\_packet\_pending = false;

switch (dwc->ep0state) {

case EP0\_SETUP\_PHASE: **dwc3\_ep0\_inspect\_setup**(dwc, event); break;

case EP0\_DATA\_PHASE: dwc3\_ep0\_complete\_data(dwc, event); break;

case EP0\_STATUS\_PHASE: dwc3\_ep0\_complete\_status(dwc, event); break;

default:

WARN(true, "UNKNOWN ep0state %d\n", dwc->ep0state);

}

}

## dwc3\_ep0\_inspect\_setup()

static void dwc3\_ep0\_inspect\_setup(struct dwc3 \*dwc, const struct dwc3\_event\_depevt \*event)

{

struct usb\_ctrlrequest \*ctrl = dwc->ctrl\_req;

len = le16\_to\_cpu(ctrl->wLength);

if (!len) {

dwc->three\_stage\_setup = false;

dwc->ep0\_expect\_in = false;

dwc->ep0\_next\_event = DWC3\_EP0\_NRDY\_STATUS;

} else {

dwc->three\_stage\_setup = true;

dwc->ep0\_expect\_in = !!(ctrl->bRequestType & USB\_DIR\_IN);

dwc->ep0\_next\_event = DWC3\_EP0\_NRDY\_DATA;

}

if ((ctrl->bRequestType & USB\_TYPE\_MASK) == USB\_TYPE\_STANDARD) ret = **dwc3\_ep0\_std\_request**(dwc, ctrl);

else ret = **dwc3\_ep0\_delegate\_req**(dwc, ctrl);

if (ret == USB\_GADGET\_DELAYED\_STATUS) dwc->delayed\_status = true;

if (ret < 0) dwc3\_ep0\_stall\_and\_restart(dwc);

}

## dwc3\_ep0\_std\_request()

**[ 481.162029] [dwc3\_ep0\_std\_request:705] hongyan ctrl->bRequest = 0x5 //USB\_REQ\_SET\_ADDRESS**

**[ 481.192060] [dwc3\_ep0\_std\_request:705] hongyan ctrl->bRequest = 0x6 //USB\_REQ\_GET\_DESCRIPTOR**

**[ 481.200998] [dwc3\_ep0\_std\_request:705] hongyan ctrl->bRequest = 0x9 //USB\_REQ\_SET\_CONFIGURATION**

**[ 481.203060] [dwc3\_ep0\_std\_request:705] hongyan ctrl->bRequest = 0xb //USB\_REQ\_SET\_INTERFACE**

static int dwc3\_ep0\_std\_request(struct dwc3 \*dwc, struct usb\_ctrlrequest \*ctrl)

{

switch (ctrl->bRequest) {

case USB\_REQ\_GET\_STATUS: ret = dwc3\_ep0\_handle\_status(dwc, ctrl); break;

case USB\_REQ\_CLEAR\_FEATURE: ret = dwc3\_ep0\_handle\_feature(dwc, ctrl, 0); break;

case USB\_REQ\_SET\_FEATURE: ret = dwc3\_ep0\_handle\_feature(dwc, ctrl, 1); break;

case USB\_REQ**\_SET\_ADDRESS**: ret = **dwc3\_ep0\_set\_address**(dwc, ctrl); break;

case USB\_REQ\_**SET\_CONFIGURATION**: ret = **dwc3\_ep0\_set\_config**(dwc, ctrl); break;

case USB\_REQ\_SET\_SEL: ret = dwc3\_ep0\_set\_sel(dwc, ctrl); break;

case USB\_REQ\_SET\_ISOCH\_DELAY: ret = dwc3\_ep0\_set\_isoch\_delay(dwc, ctrl); break;

default: ret = **dwc3\_ep0\_delegate\_req**(dwc, ctrl); break;

};

}

## dwc3\_ep0\_delegate\_req()

static int dwc3\_ep0\_delegate\_req(struct dwc3 \*dwc, struct usb\_ctrlrequest \*ctrl)

{

ret = dwc->gadget\_driver->setup(&dwc->gadget, ctrl);

}

## composite\_setup()

int composite\_setup(struct usb\_gadget \*gadget, const struct usb\_ctrlrequest \*ctrl)

{

struct usb\_composite\_dev \*cdev = get\_gadget\_data(gadget);

struct usb\_request \*req = cdev->req;

u16 w\_index = le16\_to\_cpu(ctrl->wIndex);

u8 intf = w\_index & 0xFF;

u16 w\_value = le16\_to\_cpu(ctrl->wValue);

u16 w\_length = le16\_to\_cpu(ctrl->wLength);

struct usb\_function \*f = NULL;

req->zero = 0;

req->complete = composite\_setup\_complete;

req->length = 0;

gadget->ep0->driver\_data = cdev;

switch (ctrl->bRequest) {

case USB\_REQ\_**GET\_DESCRIPTOR**:

switch (w\_value >> 8) {

case USB\_DT\_DEVICE:

cdev->desc.bNumConfigurations = count\_configs(cdev, USB\_DT\_DEVICE);

cdev->desc.bMaxPacketSize0 = cdev->gadget->ep0->maxpacket;

if (gadget\_is\_superspeed(gadget)) {

if (gadget->speed >= USB\_SPEED\_SUPER) {

cdev->desc.bcdUSB = cpu\_to\_le16(0x0300);

cdev->desc.bMaxPacketSize0 = 9;

} else { cdev->desc.bcdUSB = cpu\_to\_le16(0x0210); }

}

value = min(w\_length, (u16) sizeof cdev->desc);

memcpy(req->buf, &cdev->desc, value);

break;

case USB\_DT\_CONFIG:

value = config\_desc(cdev, w\_value);

if (value >= 0) value = min(w\_length, (u16) value);

break;

case USB\_DT\_STRING:

value = get\_string(cdev, req->buf,w\_index, w\_value & 0xff);

if (value >= 0) value = min(w\_length, (u16) value);

break;

}

break;

case USB\_REQ\_**SET\_CONFIGURATION**: value = **set\_config**(cdev, ctrl, w\_value); break;

case USB\_REQ\_**SET\_INTERFACE**:

f = cdev->config->interface[intf];

value = **f->set\_alt**(f, w\_index, w\_value); //set\_alt(1, 0)

break;

**default:**

**unknown:**

switch (ctrl->bRequestType & USB\_RECIP\_MASK) {

case USB\_RECIP\_INTERFACE: f = cdev->config->interface[intf]; break;

case USB\_RECIP\_ENDPOINT:

endp = ((w\_index & 0x80) >> 3) | (w\_index & 0x0f);

list\_for\_each\_entry(f, &cdev->config->functions, list) if (test\_bit(endp, f->endpoints)) break;

break;

}

if (f && f->setup) value = **f->setup**(f, ctrl);

else { struct usb\_configuration \*c = cdev->config; if (c && c->setup) value = c->setup(c, ctrl);}

goto done;

}

if (value >= 0 && value != USB\_GADGET\_DELAYED\_STATUS) {

req->length = value;

req->zero = value < w\_length;

value = usb\_ep\_queue(gadget->ep0, req, GFP\_ATOMIC);

if (value < 0) { req->status = 0; composite\_setup\_complete(gadget->ep0, req);

}

}

done:

return value;

}

## uvc\_function\_setup()

static int uvc\_function\_setup(struct usb\_function \*f, const struct usb\_ctrlrequest \*ctrl)

{

struct uvc\_device \*uvc = to\_uvc(f);

struct v4l2\_event v4l2\_event;

struct uvc\_event \*uvc\_event = (void \*)&v4l2\_event.u.data;

memset(&v4l2\_event, 0, sizeof(v4l2\_event));

v4l2\_event.type = UVC\_EVENT\_SETUP;

memcpy(&uvc\_event->req, ctrl, sizeof(uvc\_event->req));

v4l2\_event\_queue(uvc->vdev, &v4l2\_event);

}

## UVC\_EVENT\_SETUP

sub.type = UVC\_EVENT\_SETUP;

ioctl(dev->fd, VIDIOC\_SUBSCRIBE\_EVENT, &sub);

## =========================:

## USB\_REQ\_CLEAR\_FEATURE

### composite\_setup()

**//bRequestType=21, bRequest=01, w\_value=0100, w\_index=0001, w\_length= 26**

**//bRequestType=21, bRequest=01, w\_value=0200, w\_index=0001, w\_length= 26**

**//bRequestType = 0x21 = USB\_DIR\_OUT | USB\_TYPE\_CLASS | USB\_RECIP\_INTERFACE**

**//bRequest = 0x01 = USB\_REQ\_CLEAR\_FEATURE**

int composite\_setup(struct usb\_gadget \*gadget, const struct usb\_ctrlrequest \*ctrl)

{

switch (ctrl->bRequest) {

case **USB\_REQ\_CLEAR\_FEATURE**:

case USB\_REQ\_SET\_FEATURE:

if (ctrl->bRequestType != (USB\_DIR\_OUT | USB\_RECIP\_INTERFACE)) //(0x00 | 0x01) **goto unknown**;

break;

default:

**unknown:**

switch (ctrl->bRequestType & USB\_RECIP\_MASK) {

case USB\_RECIP\_INTERFACE: **f = cdev->config->interface[intf];**  break;

case USB\_RECIP\_ENDPOINT:

endp = ((w\_index & 0x80) >> 3) | (w\_index & 0x0f);

list\_for\_each\_entry(f, &cdev->config->functions, list) { if (test\_bit(endp, f->endpoints)) break; }

if (&f->list == &cdev->config->functions) f = NULL;

break;

}

if (f && f->setup) { **value = f->setup(f, ctrl);** // uvc\_function\_setup() }

}

}

}

### uvc\_function\_setup() : UVC\_EVENT\_SETUP

static int uvc\_function\_setup(struct usb\_function \*f, const struct usb\_ctrlrequest \*ctrl)

{

struct uvc\_device \*uvc = to\_uvc(f);

struct v4l2\_event v4l2\_event;

struct uvc\_event \*uvc\_event = (void \*)&v4l2\_event.u.data;

memset(&v4l2\_event, 0, sizeof(v4l2\_event));

v4l2\_event.type = **UVC\_EVENT\_SETUP**;

memcpy(&uvc\_event->req, ctrl, sizeof(uvc\_event->req));

v4l2\_event\_queue(uvc->vdev, &v4l2\_event);

}

## UVC\_EVENT\_SETUP : UVC\_SET\_CUR

### uvc\_gadget\_events\_process()

**//bRequestType=21, bRequest=01, w\_value=0200, w\_index=0001, w\_length= 26**

**//bRequestType = 0x21 = USB\_DIR\_OUT | USB\_TYPE\_CLASS | USB\_RECIP\_INTERFACE**

static void uvc\_gadget\_events\_process(struct uvc\_gadget \*gadget)

{

ret = **v4l2\_dequeue\_event**(gadget->out->fd, &v4l2\_event);

switch (v4l2\_event.type) {

case **UVC\_EVENT\_SETUP**:

memset(&resp, 0, sizeof(struct uvc\_request\_data));

resp.length = -EL2HLT;

**uvc\_gadget\_events\_process\_setup**(gadget, &uvc\_event->req, &resp);

uvc\_send\_response(gadget->out->fd, &resp);

break;

}

}

### v4l2\_dequeue\_event()

static inline int v4l2\_dequeue\_event(int fd, struct v4l2\_event \*event)

{

ret = xioctl(fd, **VIDIOC\_DQEVENT**, event);

}

### uvc\_v4l2\_do\_ioctl()

**//bRequestType=21, bRequest=01, w\_value=0200, w\_index=0001, w\_length= 26**

**//bRequestType = 0x21 = USB\_DIR\_OUT | USB\_TYPE\_CLASS | USB\_RECIP\_INTERFACE**

static long uvc\_v4l2\_do\_ioctl(struct file \*file, unsigned int cmd, void \*arg)

{

case VIDIOC\_**DQEVENT**:

{

struct v4l2\_event \*event = arg;

ret = **v4l2\_event\_dequeue**(&handle->vfh, event,file->f\_flags & O\_NONBLOCK);

if (ret == 0 && **event->type == UVC\_EVENT\_SETUP**) {

struct uvc\_event \*uvc\_event = (void \*)&event->u.data;

**uvc->event\_setup\_out = !(uvc\_event->req.bRequestType & USB\_DIR\_IN);** **//bRequestType: b[7]=USB\_DIR\_OUT=0**

uvc->event\_length = uvc\_event->req.wLength;

}

return ret;

}

}

### uvc\_function\_ep0\_complete():UVC\_EVENT\_DATA

static void uvc\_function\_ep0\_complete(struct usb\_ep \*ep, struct usb\_request \*req)

{

struct uvc\_device \*uvc = req->context;

struct v4l2\_event v4l2\_event;

struct uvc\_event \*uvc\_event = (void \*)&v4l2\_event.u.data;

**if (uvc->event\_setup\_out)** {

uvc->event\_setup\_out = 0;

memset(&v4l2\_event, 0, sizeof(v4l2\_event));

v4l2\_event.type = **UVC\_EVENT\_DATA**;

uvc\_event->data.length = req->actual;

memcpy(&uvc\_event->data.data, req->buf, req->actual);

v4l2\_event\_queue(uvc->vdev, &v4l2\_event);

}

}

### uvc\_gadget\_events\_process\_setup()

**//bRequestType=21, bRequest=01, w\_value=0200, w\_index=0001, w\_length= 26**

static void uvc\_gadget\_events\_process\_setup(struct uvc\_gadget \*gadget,struct usb\_ctrlrequest \*ctrl,struct uvc\_request\_data \*resp)

{

switch (ctrl->bRequestType & USB\_TYPE\_MASK) {

case **USB\_TYPE\_CLASS**: **uvc\_gadget\_events\_process\_class**(gadget, ctrl, resp); break;

}

}

static void uvc\_gadget\_events\_process\_class(struct uvc\_gadget \*gadget,struct usb\_ctrlrequest \*ctrl,struct uvc\_request\_data \*resp)

{

switch (ctrl->wIndex & 0xff) {

case **UVC\_INTF\_STREAMING: uvc\_gadget\_events\_process\_streaming**(gadget, ctrl->bRequest,ctrl->wValue >> 8, resp);break;

}

}

### uvc\_gadget\_events\_process\_streaming()

**//bRequestType=21, bRequest=01, w\_value=0200, w\_index=0001, w\_length= 26**

**//bRequestType = 0x21 = USB\_DIR\_OUT | USB\_TYPE\_CLASS | USB\_RECIP\_INTERFACE**

**// bRequest=0x01 =** **UVC\_SET\_CUR**

**// wIndex=0x0001= UVC\_INTF\_STREAMING**

**// cs = w\_value>>8 = 0x0200>>8 = 0x02 = UVC\_VS\_COMMIT\_CONTROL**

static void uvc\_gadget\_events\_process\_streaming(struct uvc\_gadget \*gadget, uint8\_t req, uint8\_t cs, struct uvc\_request\_data \*resp)

{

switch (req) {

case UVC\_SET\_CUR: gadget->control = cs; break;

}

}

## UVC\_EVENT\_DATA: COMMIT

### uvc\_gadget\_events\_process()

**//bRequestType=21, bRequest=01, w\_value=0200, w\_index=0001, w\_length= 26**

**//bRequestType = 0x21 = USB\_DIR\_OUT | USB\_TYPE\_CLASS | USB\_RECIP\_INTERFACE**

static void uvc\_gadget\_events\_process(struct uvc\_gadget \*gadget)

{

ret = **v4l2\_dequeue\_event**(gadget->out->fd, &v4l2\_event);

switch (v4l2\_event.type) {

case **UVC\_EVENT\_DATA**: uvc\_gadget\_events\_process\_data(gadget, &uvc\_event->data); break;

}

}

### uvc\_gadget\_events\_process\_data()

**//bRequestType=21, bRequest=01, w\_value=0200, w\_index=0001, w\_length= 26**

**//bRequestType = 0x21 = USB\_DIR\_OUT | USB\_TYPE\_CLASS | USB\_RECIP\_INTERFACE**

**// bRequest=0x01 =** **UVC\_SET\_CUR**

**// wIndex=0x0001= UVC\_INTF\_STREAMING**

**// cs = w\_value>>8 = 0x0200>>8 = 0x02 = UVC\_VS\_COMMIT\_CONTROL**

static void uvc\_gadget\_events\_process\_data(struct uvc\_gadget \*gadget, struct uvc\_request\_data \*data)

{

struct uvc\_streaming\_control \*target;

struct uvc\_streaming\_control \*ctrl;

switch (gadget->control) {

case UVC\_VS\_PROBE\_CONTROL: target = &gadget->probe; break;

case **UVC\_VS\_COMMIT\_CONTRO**L: **target = &gadget->commit**; break;

}

ctrl = (struct uvc\_streaming\_control \*)&data->data;

iformat = CLAMP((unsigned int)ctrl->bFormatIndex, 1U, (unsigned int)ARRAY\_SIZE(uvc\_gadget\_formats));

**format = &uvc\_gadget\_formats[iformat - 1];**

nframes = 0;

while (format->frames[nframes].width != 0) nframes++;

iframe = CLAMP((unsigned int)ctrl->bFrameIndex, (unsigned int)1U, nframes);

**frame = &format->frames[iframe - 1];**

interval = frame->intervals;

while (interval[0] < ctrl->dwFrameInterval && interval[1]) interval++;

**target->bFormatIndex = iformat;**

**target->bFrameIndex = iframe;**

switch (format->fcc) {

case V4L2\_PIX\_FMT\_YUYV: target->dwMaxVideoFrameSize = frame->width \* frame->height \* 2; break;

}

target->dwFrameInterval = \*interval;

if (gadget->control == **UVC\_VS\_COMMIT\_CONTROL**) {

**gadget->out->fcc = format->fcc;**

**gadget->out->width = frame->width;**

**gadget->out->height = frame->height;**

**uvc\_gadget\_set\_format**(gadget->out,(format->fcc == V4L2\_PIX\_FMT\_MJPEG) ?gadget->imgsize : 0);

}

}

#### struct uvc\_gadget\_format\_info uvc\_gadget\_formats

static const struct uvc\_gadget\_format\_info uvc\_gadget\_formats[] = {

{ V4L2\_PIX\_FMT\_YUYV, uvc\_gadget\_frames\_yuyv },

//{ V4L2\_PIX\_FMT\_NV12, uvc\_gadget\_frames\_nv12 },

{ V4L2\_PIX\_FMT\_MJPEG, uvc\_gadget\_frames\_mjpg },

};

#### struct uvc\_gadget\_frame\_info uvc\_gadget\_frames\_yuyv

static const struct uvc\_gadget\_frame\_info uvc\_gadget\_frames\_yuyv[] = {

{ 176, 144, { 666666, 1000000, 5000000, 0 }, },

{ 640, 360, { 666666, 1000000, 5000000, 0 }, },

{ 1280, 720, { 50000000, 0 }, },

{ 0, 0, { 0, }, },

};

## USB\_REQ\_SET\_INTERFACE

### composite\_setup()

**// bRequestType=01, bRequest=0b, w\_value=0001, w\_index=0001, w\_length= 0**

int composite\_setup(struct usb\_gadget \*gadget, const struct usb\_ctrlrequest \*ctrl)

{

case USB\_REQ\_SET\_INTERFACE:

f = cdev->config->interface[intf];

value = **f->set\_alt**(f, w\_index, w\_value); // set\_alt(1, 1) : uvc\_function\_set\_alt()

if (value == USB\_GADGET\_DELAYED\_STATUS) {

cdev->delayed\_status++;

}

break;

}

### uvc\_function\_set\_alt()

static int uvc\_function\_set\_alt(struct usb\_function \*f, unsigned interface, unsigned alt)

{

struct uvc\_device \*uvc = to\_uvc(f);

struct v4l2\_event v4l2\_event;

struct uvc\_event \*uvc\_event = (void \*)&v4l2\_event.u.data;

switch (alt) {

**case 1:**

if (uvc->video.ep) {

ret = **config\_ep\_by\_speed**(f->config->cdev->gadget,&(uvc->func), uvc->video.ep);

**usb\_ep\_enable**(uvc->video.ep);

}

memset(&v4l2\_event, 0, sizeof(v4l2\_event));

**v4l2\_event.type = UVC\_EVENT\_STREAMON;**

v4l2\_event\_queue(uvc->vdev, &v4l2\_event);

return USB\_GADGET\_DELAYED\_STATUS;

}

}

## UVC\_EVENT\_STREAMON:

### uvc\_gadget\_events\_process()

static void uvc\_gadget\_events\_process(struct uvc\_gadget \*gadget)

{

ret = v4l2\_dequeue\_event(gadget->out->fd, &v4l2\_event);

switch (v4l2\_event.type) {

case **UVC\_EVENT\_STREAMON:**

**uvc\_gadget\_reqbufs**(gadget->out, NR\_VIDEO\_BUF); //#define NR\_VIDEO\_BUF (4)

uvc\_gadget\_stream(gadget->out, VIDEO\_STREAM\_ON);

break;

}

}

## VIDIOC\_REQBUFS

### uvc\_gadget\_reqbufs()

static int uvc\_gadget\_reqbufs(struct uvc\_gadget\_device \*dev, unsigned **int nbufs)**

{

struct v4l2\_requestbuffers rbuf;

memset(&rbuf, 0, sizeof(struct v4l2\_requestbuffers));

rbuf.count = nbufs;

rbuf.type = dev->type;

rbuf.memory = **V4L2\_MEMORY\_MMAP**;

ret = **v4l2\_request\_buffer**(dev->fd, &rbuf);

**dev->buf** = malloc(rbuf.count \* sizeof(void \*));

}

static inline int v4l2\_request\_buffer(int fd, struct v4l2\_requestbuffers \*rbuf)

{

ret = xioctl(fd, **VIDIOC\_REQBUFS**, rbuf);

}

### uvc\_v4l2\_do\_ioctl()

static long uvc\_v4l2\_do\_ioctl(struct file \*file, unsigned int cmd, void \*arg)

{

case **VIDIOC\_REQBUFS**:

{

struct v4l2\_requestbuffers \*rb = arg;

ret = **uvc\_alloc\_buffers**(&video->queue, rb);

break;

}

}

### uvc\_alloc\_buffers() //初始化vb2\_queue

static int uvc\_alloc\_buffers(**struct uvc\_video\_queue** \*queue, struct v4l2\_requestbuffers \*rb)

{

ret = **vb2\_reqbufs**(&queue->queue, rb);

**return ret ? ret : rb->count;**

}

int vb2\_reqbufs(**struct vb2\_queue** \*q, struct v4l2\_requestbuffers \*req)

{

int ret = \_\_verify\_memory\_type(q, req->memory, req->type);

return ret ? ret : **\_\_reqbufs**(q, req);

}

static int \_\_reqbufs(**struct vb2\_queue** \*q, struct v4l2\_requestbuffers \*req)

{

unsigned int num\_planes = 0;

num\_buffers = min\_t(unsigned int, req->count, VIDEO\_MAX\_FRAME);

memset(q->plane\_sizes, 0, sizeof(q->plane\_sizes));

memset(q->alloc\_ctx, 0, sizeof(q->alloc\_ctx));

**q->memory** = req->memory;

ret = **call\_qop**(q, **queue\_setup**, q, NULL, **&num\_buffers**, **&num\_planes**, **q->plane\_sizes**, q->alloc\_ctx);

ret = **\_\_vb2\_queue\_alloc**(q, req->memory, **num\_buffers**, **num\_planes**); // **num\_buffers =4, num\_planes =1**

allocated\_buffers = ret;

**q->num\_buffers** = allocated\_buffers;

**req->count** = allocated\_buffers;

}

### uvc\_queue\_setup()

static int uvc\_queue\_setup(**struct vb2\_queue** \*vq, const struct v4l2\_format \*fmt,

unsigned int \*nbuffers, unsigned int \*nplanes, unsigned int sizes[], void \*alloc\_ctxs[])

{

**struct uvc\_video\_queue** \*queue = vb2\_get\_drv\_priv(vq);

**struct uvc\_video** \*video = container\_of(queue, struct uvc\_video, queue);

if (\*nbuffers > UVC\_MAX\_VIDEO\_BUFFERS) \*nbuffers = UVC\_MAX\_VIDEO\_BUFFERS;

**\*nplanes = 1;**

**sizes[0] = video->imagesize;**

}

### \_\_vb2\_queue\_alloc() //初始化vb2\_buffer, v4l2\_buffer

static int \_\_vb2\_queue\_alloc(struct vb2\_queue \*q, enum v4l2\_memory memory, unsigned int num\_buffers, unsigned int num\_planes)

{

unsigned int buffer;

for (buffer = 0; buffer < num\_buffers; ++buffer) {

**struct vb2\_buffer** \*vb = kzalloc(q->buf\_struct\_size, GFP\_KERNEL);

**vb->state = VB2\_BUF\_STATE\_DEQUEUED;**

vb->vb2\_queue = q;

vb->**num\_planes = num\_planes;**

vb->v4l2\_buf.index = q->num\_buffers + buffer;

vb->v4l2\_buf.type = q->type;

vb->v4l2\_buf.memory = memory;

if (memory == V4L2\_MEMORY\_MMAP) {

ret = **\_\_vb2\_buf\_mem\_alloc**(vb);

ret = call\_qop(q, **buf\_init**, vb); //struct vb2\_ops uvc\_queue\_qops: not init

}

q->bufs[q->num\_buffers + buffer] = vb;

}

\_\_setup\_offsets(q, buffer);

**return buffer;**

}

### \_\_vb2\_buf\_mem\_alloc() //初始化vb2\_plane, v4l2\_plane

static int \_\_vb2\_buf\_mem\_alloc(struct vb2\_buffer \*vb)

{

struct vb2\_queue \*q = vb->vb2\_queue;

for (plane = 0; plane < vb->num\_planes; ++plane) {

unsigned long size = PAGE\_ALIGN(q->plane\_sizes[plane]); // **video->imagesize**

mem\_priv = call\_memop(q, **alloc**, q->alloc\_ctx[plane], size, q->gfp\_flags); //vb2\_vmalloc\_alloc()

**vb->planes[plane].mem\_priv** = mem\_priv;

**vb->v4l2\_planes[plane].length = q->plane\_sizes[plane];**

}

}

## VIDIOC\_QBUF:

### uvc\_gadget\_reqbufs()

static int uvc\_gadget\_reqbufs(struct uvc\_gadget\_device \*dev, unsigned **int nbufs)**

{

for (i = 0; i < rbuf.count; i++) {

memset(&dev->v4l2\_buf, 0, sizeof(struct v4l2\_buffer));

dev->v4l2\_buf.index = i;

dev->v4l2\_buf.type = dev->type;

dev->v4l2\_buf.memory = V4L2\_MEMORY\_MMAP;

ret = **v4l2\_query\_buffer**(dev->fd, &dev->v4l2\_buf);

**dev->buf[i]** = **mmap**(0, dev->v4l2\_buf.length,PROT\_READ | PROT\_WRITE, MAP\_SHARED, dev->fd, dev->v4l2\_buf.m.offset);

dev->v4l2\_buf.index = i;

dev->v4l2\_buf.type = dev->type;

dev->v4l2\_buf.memory = V4L2\_MEMORY\_MMAP;

ret = **v4l2\_queue\_buf(**dev->fd, &dev->v4l2\_buf);

}

**dev->bufsize** = dev->v4l2\_buf.length;

**dev->nbufs** = rbuf.count;

}

### v4l2\_query\_buffer()

static inline int v4l2\_query\_buffer(int fd, struct v4l2\_buffer \*buf)

{

ret = xioctl(fd, **VIDIOC\_QUERYBUF**, buf);

}

static void \_\_fill\_v4l2\_buffer(struct vb2\_buffer \*vb, struct v4l2\_buffer \*b)

{

struct vb2\_queue \*q = vb->vb2\_queue;

memcpy(b, &vb->v4l2\_buf, offsetof(struct v4l2\_buffer, m));

b->reserved2 = vb->v4l2\_buf.reserved2;

b->reserved = vb->v4l2\_buf.reserved;

if (V4L2\_TYPE\_IS\_MULTIPLANAR(q->type)) {

} else {

**b->length = vb->v4l2\_planes[0].length;**

**b->bytesused = vb->v4l2\_planes[0].bytesused;**

if ( q->memory == V4L2\_MEMORY\_MMAP) **b->m.offset = vb->v4l2\_planes[0].m.mem\_offset;**

else if (q->memory == V4L2\_MEMORY\_USERPTR) b->m.userptr = vb->v4l2\_planes[0].m.userptr;

else if (q->memory == V4L2\_MEMORY\_DMABUF) b->m.fd = vb->v4l2\_planes[0].m.fd;

}

}

### v4l2\_queue\_buf()

static inline int v4l2\_queue\_buf(int fd, struct v4l2\_buffer \*buf)

{

ret = xioctl(fd, **VIDIOC\_QBUF,** buf);

}

### uvc\_v4l2\_do\_ioctl()

static long uvc\_v4l2\_do\_ioctl(struct file \*file, unsigned int cmd, void \*arg)

{

case **VIDIOC\_QBUF:**

ret = **uvc\_queue\_buffer**(&video->queue, arg);

return **uvc\_video\_pump**(video);

break;

}

### uvc\_queue\_buffer():list\_add\_tail(&vb->queued\_entry, &q->queued\_list)

static int uvc\_queue\_buffer(struct uvc\_video\_queue \*queue, struct v4l2\_buffer \*buf)

{

ret = **vb2\_qbuf**(&queue->queue, buf);

queue->flags &= ~UVC\_QUEUE\_PAUSED;

}

int vb2\_qbuf(struct vb2\_queue \*q, struct v4l2\_buffer \*b)

{

struct vb2\_buffer \*vb = q->bufs[b->index];

ret = \_\_verify\_planes\_array(vb, b);

switch (vb->state) {

case VB2\_BUF\_STATE\_DEQUEUED: ret = **\_\_buf\_prepare**(vb, b);

case VB2\_BUF\_STATE\_PREPARED: break;

}

**list\_add\_tail(&vb->queued\_entry, &q->queued\_list);**

**vb->state = VB2\_BUF\_STATE\_QUEUED;**

if (q->streaming) **\_\_enqueue\_in\_driver**(vb);

**\_\_fill\_v4l2\_buffer**(vb, b);

}

### \_\_buf\_prepare()

static int \_\_buf\_prepare(struct vb2\_buffer \*vb, const struct v4l2\_buffer \*b)

{

struct vb2\_queue \*q = vb->vb2\_queue;

switch (q->memory) {

case V4L2\_MEMORY\_MMAP: ret = **\_\_qbuf\_mmap**(vb, b); break;

case V4L2\_MEMORY\_USERPTR: ret = \_\_qbuf\_userptr(vb, b); break;

case V4L2\_MEMORY\_DMABUF: ret = \_\_qbuf\_dmabuf(vb, b); break;

}

if (!ret) ret = call\_qop(q, **buf\_prepare**, vb);

vb->state = VB2\_BUF\_STATE\_PREPARED;

}

### \_\_qbuf\_mmap() //初始化 v4l2\_plane, v4l2\_buffer

static int \_\_qbuf\_mmap(struct vb2\_buffer \*vb, const struct v4l2\_buffer \*b)

{

**\_\_fill\_vb2\_buffer**(vb, b, vb->v4l2\_planes);

}

static void \_\_fill\_vb2\_buffer(struct vb2\_buffer \*vb, const struct v4l2\_buffer \*b, struct v4l2\_plane \*v4l2\_planes)

{

if (V4L2\_TYPE\_IS\_MULTIPLANAR(b->type)) {

} else {

if (V4L2\_TYPE\_IS\_OUTPUT(b->type)) { **v4l2\_planes[0].bytesused = b->bytesused; v4l2\_planes[0].data\_offset = 0;** }

if (V4L2\_MEMORY\_USERPTR) { v4l2\_planes[0].m.userptr = b->m.userptr; v4l2\_planes[0].length = b->length; }

if (V4L2\_MEMORY\_DMABUF) { v4l2\_planes[0].m.fd = b->m.fd; v4l2\_planes[0].length = b->length; v4l2\_planes[0].data\_offset = 0; }

}

**vb->v4l2\_buf.field = b->field;**

**vb->v4l2\_buf.timestamp = b->timestamp;**

vb->v4l2\_buf.flags = b->flags & ~V4L2\_BUFFER\_MASK\_FLAGS;

}

### uvc\_buffer\_prepare() //初始化uvc\_buffer

static int uvc\_buffer\_prepare(struct vb2\_buffer \*vb)

{

struct uvc\_video\_queue \*queue = vb2\_get\_drv\_priv(vb->vb2\_queue);

**struct uvc\_buffer** \*buf = container\_of(vb, struct uvc\_buffer, buf);

**buf->state = UVC\_BUF\_STATE\_QUEUED;**

**buf->mem = vb2\_plane\_vaddr(vb, 0);**

**buf->length = vb2\_plane\_size(vb, 0);**

**buf->bytesused = vb2\_get\_plane\_payload(vb, 0);**

}

void \*vb2\_plane\_vaddr(struct vb2\_buffer \*vb, unsigned int plane\_no)

{

return call\_memop(q, vaddr, **vb->planes[plane\_no].mem\_priv**); //vb2\_vmalloc\_vaddr()

}

static inline unsigned long vb2\_plane\_size(struct vb2\_buffer \*vb, unsigned int plane\_no)

{

return **vb->v4l2\_planes[plane\_no].length**;

}

static inline unsigned long vb2\_get\_plane\_payload(struct vb2\_buffer \*vb, unsigned int plane\_no)

{

return **vb->v4l2\_planes[plane\_no].bytesused**;

}

### \_\_enqueue\_in\_driver() // list\_add\_tail(&buf->queue, &queue->irqqueue)

static void \_\_enqueue\_in\_driver(struct vb2\_buffer \*vb)

{

vb->state = VB2\_BUF\_STATE\_ACTIVE;

atomic\_inc(&q->queued\_count);

for (plane = 0; plane < vb->num\_planes; ++plane)

call\_memop(q, prepare, vb->planes[plane].mem\_priv); // struct vb2\_mem\_ops vb2\_vmalloc\_memops: 没定义

q->ops->buf\_queue(vb); //uvc\_buffer\_queue()

}

static void uvc\_buffer\_queue(struct vb2\_buffer \*vb)

{

struct uvc\_video\_queue \*queue = vb2\_get\_drv\_priv(vb->vb2\_queue);

struct uvc\_buffer \*buf = container\_of(vb, struct uvc\_buffer, buf);

if (likely(!(queue->flags & UVC\_QUEUE\_DISCONNECTED))) {

**list\_add\_tail(&buf->queue, &queue->irqqueue);**

} else {

buf->state = UVC\_BUF\_STATE\_ERROR;

vb2\_buffer\_done(&buf->buf, VB2\_BUF\_STATE\_ERROR);

}

}

### \_\_fill\_v4l2\_buffer() //初始化v4l2\_buffer

static void \_\_fill\_v4l2\_buffer(struct vb2\_buffer \*vb, struct v4l2\_buffer \*b)

{

struct vb2\_queue \*q = vb->vb2\_queue;

memcpy(b, &vb->v4l2\_buf, offsetof(struct v4l2\_buffer, m));

b->reserved2 = vb->v4l2\_buf.reserved2;

b->reserved = vb->v4l2\_buf.reserved;

if (V4L2\_TYPE\_IS\_MULTIPLANAR(q->type)) {

} else {

**b->length = vb->v4l2\_planes[0].length;**

**b->bytesused = vb->v4l2\_planes[0].bytesused;**

if ( q->memory == V4L2\_MEMORY\_MMAP) **b->m.offset = vb->v4l2\_planes[0].m.mem\_offset;**

else if (q->memory == V4L2\_MEMORY\_USERPTR) b->m.userptr = vb->v4l2\_planes[0].m.userptr;

else if (q->memory == V4L2\_MEMORY\_DMABUF) b->m.fd = vb->v4l2\_planes[0].m.fd;

}

}

## VIDIOC\_STREAMON: list\_add\_tail(&video->req[i]->list, &video->req\_free)

### uvc\_gadget\_events\_process()

static void uvc\_gadget\_events\_process(struct uvc\_gadget \*gadget)

{

ret = v4l2\_dequeue\_event(gadget->out->fd, &v4l2\_event);

switch (v4l2\_event.type) {

case UVC\_EVENT\_STREAMON:

uvc\_gadget\_reqbufs(gadget->out, NR\_VIDEO\_BUF); //#define NR\_VIDEO\_BUF (4)

**uvc\_gadget\_stream**(gadget->out, **VIDEO\_STREAM\_ON**);

break;

}

}

static int uvc\_gadget\_stream(struct uvc\_gadget\_device \*dev, int enable)

{

if (enable) ret = **v4l2\_stream\_on**(dev->fd, dev->type);

}

static inline int v4l2\_stream\_on(int fd, int type)

{

ret = xioctl(fd, **VIDIOC\_STREAMON**, &type);

}

### uvc\_v4l2\_do\_ioctl()

uvc\_v4l2\_do\_ioctl(struct file \*file, unsigned int cmd, void \*arg)

{

**case VIDIOC\_STREAMON:**

{

ret = **uvc\_video\_enable**(video, 1);

uvc\_function\_setup\_continue(uvc);

uvc->state = UVC\_STATE\_STREAMING;

}

}

### uvc\_video\_enable()

static int uvc\_video\_enable(struct uvc\_video \*video, int enable)

{

ret = **uvc\_queue\_enable**(&video->queue, 1);

ret = **uvc\_video\_alloc\_requests**(video);

if (video->max\_payload\_size) {

video->encode = uvc\_video\_encode\_bulk;

video->payload\_size = 0;

} else

video->encode = **uvc\_video\_encode\_isoc**;

return **uvc\_video\_pump**(video);

}

### uvc\_queue\_enable()

static int uvc\_queue\_enable(struct uvc\_video\_queue \*queue, int enable)

{

if (enable) {

ret = **vb2\_streamon**(&queue->queue, queue->queue.type);

queue->sequence = 0;

**queue->buf\_used = 0;**

}

}

int vb2\_streamon(struct vb2\_queue \*q, enum v4l2\_buf\_type type)

{

call\_qop(q, start\_streaming, q, atomic\_read(&q->queued\_count)); // struct vb2\_ops uvc\_queue\_qops没有定义start\_streaming()

**q->streaming = 1;**

}

### uvc\_video\_alloc\_requests() // list\_add\_tail(&video->req[i]->list, &video->req\_free);

static int uvc\_video\_alloc\_requests(struct uvc\_video \*video)

{

req\_size = video->ep->maxpacket \* max\_t(unsigned int, video->ep->maxburst, 1) \* (video->ep->mult + 1);

for (i = 0; i < UVC\_NUM\_REQUESTS; ++i) { //#define UVC\_NUM\_REQUESTS 4

**video->req\_buffer[i] = kmalloc**(req\_size, GFP\_KERNEL);

video->req[i] = **usb\_ep\_alloc\_request**(video->ep, GFP\_KERNEL); //初始化usb\_request

video->req[i]->buf = video->req\_buffer[i];

video->req[i]->length = 0;

video->req[i]->complete = uvc\_video\_complete;

video->req[i]->context = video;

**list\_add\_tail(&video->req[i]->list, &video->req\_free);**

}

video->req\_size = req\_size;

}

static inline struct usb\_request \*usb\_ep\_alloc\_request(struct usb\_ep \*ep, gfp\_t gfp\_flags)

{

return ep->ops->alloc\_request(ep, gfp\_flags); // dwc3\_gadget\_ep\_alloc\_request()

}

static struct usb\_request \*dwc3\_gadget\_ep\_alloc\_request(struct usb\_ep \*ep, gfp\_t gfp\_flags)

{

struct dwc3\_ep \*dep = to\_dwc3\_ep(ep);

**struct dwc3\_request \*req = kzalloc**(sizeof(\*req), gfp\_flags);

req->epnum = dep->number;

req->dep = dep;

**return &req->request;**

}

## Video data encode

### uvc\_video\_pump()

static int **uvc\_video\_pump**(struct uvc\_video \*video)

{

while (1) {

if (list\_empty(&video->req\_free)) return 0;

struct usb\_request \*req = list\_first\_entry(&video->req\_free, struct usb\_request, list);

list\_del(&req->list);

struct uvc\_buffer \*buf = **uvc\_queue\_head**(&video->queue);

if (buf == NULL) { break; }

**video->encode**(req, video, buf); // uvc\_video\_encode\_isoc()

ret = **usb\_ep\_queue**(video->ep, req, GFP\_ATOMIC);

if (ret < 0) { usb\_ep\_set\_halt(video->ep); break; }

}

list\_add\_tail(&req->list, &video->req\_free);

}

static struct uvc\_buffer \***uvc\_queue\_head**(struct uvc\_video\_queue \*queue)

{

struct uvc\_buffer \*buf = NULL;

if (!list\_empty(&queue->irqqueue)) **buf = list\_first\_entry(&queue->irqqueue, struct uvc\_buffer,queue);**

else queue->flags |= UVC\_QUEUE\_PAUSED;

return buf;

}

### uvc\_video\_encode\_isoc()

static void uvc\_video\_encode\_isoc(**struct usb\_request** \*req, struct uvc\_video \*video, struct uvc\_buffer \*buf)

{

void \*mem = **req->buf;**

int len = video->req\_size; // len = 1024

ret = **uvc\_video\_encode\_header**(video, buf, mem, len); //ret = 2: header size

mem += ret; len -= ret;

ret = **uvc\_video\_encode\_data**(video, buf, mem, len);

len -= ret;

**req->length** = video->req\_size - len;

if (buf->bytesused == video->queue.buf\_used) {

video->queue.buf\_used = 0;

buf->state = UVC\_BUF\_STATE\_DONE;

**uvc\_queue\_next\_buffer**(&video->queue, buf);

**video->fid ^= UVC\_STREAM\_FID;**

}

}

### uvc\_video\_encode\_header()

**#define UVC\_STREAM\_EOH (1 << 7)**

**#define UVC\_STREAM\_ERR (1 << 6)**

**#define UVC\_STREAM\_STI (1 << 5)**

**#define UVC\_STREAM\_RES (1 << 4)**

**#define UVC\_STREAM\_SCR (1 << 3)**

**#define UVC\_STREAM\_PTS (1 << 2)**

**#define UVC\_STREAM\_EOF (1 << 1)**

**#define UVC\_STREAM\_FID (1 << 0)**

**//data: 0x02 0x80 or 0x02 0x81 : UVC\_STREAM\_EOH | UVC\_STREAM\_FID**

**//data: 0x02 0x82 or 0x02 0x83 : UVC\_STREAM\_EOH | UVC\_STREAM\_FID | UVC\_STREAM\_EOF**

static int **uvc\_video\_encode\_header**(struct uvc\_video \*video, struct uvc\_buffer \*buf, u8 \*data, int len)

{

data[0] = 2; //header size

data[1] = **UVC\_STREAM\_EOH** | **video->fid**; // video->fid ^= UVC\_STREAM\_FID;

if (buf->bytesused - video->queue.buf\_used <= len - 2) data[1] |= **UVC\_STREAM\_EOF**;

return 2;

}

### uvc\_video\_encode\_data()

static int **uvc\_video\_encode\_data**(struct uvc\_video \*video, struct uvc\_buffer \*buf, u8 \*data, int len)

{

struct uvc\_video\_queue \*queue = &video->queue;

void \*mem = buf->mem + queue->buf\_used;

nbytes = min((unsigned int)len, buf->bytesused - queue->buf\_used);

memcpy(data, mem, nbytes); // nbytes = 1024 - 2

**queue->buf\_used += nbytes;**

return nbytes;

}

### uvc\_queue\_next\_buffer()

static struct uvc\_buffer \*uvc\_queue\_next\_buffer(struct uvc\_video\_queue \*queue, struct uvc\_buffer \*buf)

{

struct uvc\_buffer \*nextbuf;

if ((queue->flags & UVC\_QUEUE\_DROP\_INCOMPLETE) && buf->length != buf->bytesused) {

buf->state = UVC\_BUF\_STATE\_QUEUED;

vb2\_set\_plane\_payload(&buf->buf, 0, 0);

return buf;

}

**list\_del(&buf->queue);**

if (!list\_empty(&queue->irqqueue)) **nextbuf = list\_first\_entry(&queue->irqqueue, struct uvc\_buffer, queue);**

else nextbuf = NULL;

buf->buf.v4l2\_buf.sequence = queue->sequence++;

do\_gettimeofday(&buf->buf.v4l2\_buf.timestamp);

vb2\_set\_plane\_payload(&buf->buf, 0, buf->bytesused); // vb->v4l2\_planes[plane\_no].bytesused = buf->bytesused

**vb2\_buffer\_done**(&buf->buf, VB2\_BUF\_STATE\_DONE);

return nextbuf;

}

### vb2\_buffer\_done()

void vb2\_buffer\_done(struct vb2\_buffer \*vb, enum vb2\_buffer\_state state)

{

struct vb2\_queue \*q = vb->vb2\_queue;

for (plane = 0; plane < vb->num\_planes; ++plane)

call\_memop(q, finish, vb->planes[plane].mem\_priv); // struct vb2\_mem\_ops vb2\_vmalloc\_memops: 没定义

vb->state = state;

**list\_add\_tail(&vb->done\_entry, &q->done\_list);**

atomic\_dec(&q->queued\_count);

wake\_up(&q->done\_wq);

}

### usb\_ep\_queue()

static inline int usb\_ep\_queue(struct usb\_ep \*ep, struct usb\_request \*req, gfp\_t gfp\_flags)

{

return ep->ops->queue(ep, req, gfp\_flags); // dwc3\_gadget\_ep\_queue()

}

## USB data queue

### dwc3\_gadget\_ep\_queue() // list\_add\_tail(&req->list, &dep->request\_list);

static int dwc3\_gadget\_ep\_queue(struct usb\_ep \*ep, struct usb\_request \*request, gfp\_t gfp\_flags)

{

struct dwc3\_request \*req = to\_dwc3\_request(request);

struct dwc3\_ep \*dep = to\_dwc3\_ep(ep);

ret = **\_\_dwc3\_gadget\_ep\_queue**(dep, req);

}

static int **\_\_dwc3\_gadget\_ep\_queue**(struct dwc3\_ep \*dep, struct dwc3\_request \*req)

{

struct dwc3 \*dwc = dep->dwc;

**req->request.actual = 0;**

req->request.status = -EINPROGRESS;

req->direction = dep->direction;

req->epnum = dep->number;

ret = **usb\_gadget\_map\_request**(&dwc->gadget, &req->request,dep->direction);

**list\_add\_tail(&req->list, &dep->request\_list);**

if (dep->flags & DWC3\_EP\_PENDING\_REQUEST) {

if (usb\_endpoint\_xfer\_isoc(dep->endpoint.desc)) {

if (list\_empty(&dep->req\_queued)) {

dwc3\_stop\_active\_transfer(dwc, dep->number);

dep->flags = DWC3\_EP\_ENABLED;

}

return 0;

}

ret = \_\_dwc3\_gadget\_kick\_transfer(dep, 0, true);

return ret;

}

if (usb\_endpoint\_xfer\_isoc(dep->endpoint.desc) &&(dep->**flags & DWC3\_EP\_BUSY**) &&!(dep->**flags & DWC3\_EP\_MISSED\_ISOC**)) {

ret = **\_\_dwc3\_gadget\_kick\_transfer**(dep, dep->resource\_index,false);

return ret;

}

return 0;

}

### usb\_gadget\_map\_request()

int usb\_gadget\_map\_request(struct usb\_gadget \*gadget, struct usb\_request \*req, int is\_in)

{

if (req->num\_sgs) {

mapped = dma\_map\_sg(&gadget->dev, req->sg, req->num\_sgs, is\_in ? DMA\_TO\_DEVICE : DMA\_FROM\_DEVICE);

req->num\_mapped\_sgs = mapped;

} else {

req->dma = dma\_map\_single(&gadget->dev, req->buf, req->length, is\_in ? DMA\_TO\_DEVICE : DMA\_FROM\_DEVICE);

}

}

### \_\_dwc3\_gadget\_kick\_transfer() // start\_new = 0

static int \_\_dwc3\_gadget\_kick\_transfer(struct dwc3\_ep \*dep, u16 cmd\_param, int start\_new)

{

struct dwc3\_request \*req;

dep->**flags &= ~DWC3\_EP\_PENDING\_REQUEST**;

if (start\_new) {

if (list\_empty(&dep->req\_queued)) dwc3\_prepare\_trbs(dep, start\_new);

req = next\_request(&dep->req\_queued);

} else {

**dwc3\_prepare\_trbs**(dep, start\_new);

req = **next\_request**(&dep->req\_queued);

}

struct dwc3\_gadget\_ep\_cmd\_params params;

memset(&params, 0, sizeof(params));

if (start\_new) {

params.param0 = upper\_32\_bits(req->trb\_dma);

params.param1 = lower\_32\_bits(req->trb\_dma);

cmd = DWC3\_DEPCMD\_STARTTRANSFER;

} else { cmd = **DWC3\_DEPCMD\_UPDATETRANSFER**; }

cmd |= DWC3\_DEPCMD\_PARAM(cmd\_param);

ret = **dwc3\_send\_gadget\_ep\_cmd**(dwc, dep->number, cmd, &params);

dep->**flags |= DWC3\_EP\_BUSY**;

if (start\_new) {

dep->resource\_index = dwc3\_gadget\_ep\_get\_transfer\_index(dwc,dep->number);

WARN\_ON\_ONCE(!dep->resource\_index);

}

}

### dwc3\_prepare\_trbs()

static void dwc3\_prepare\_trbs(struct dwc3\_ep \*dep, bool starting)

{

struct dwc3\_request \*req, \*n;

trbs\_left = (dep->busy\_slot - dep->free\_slot) & DWC3\_TRB\_MASK;

if (!trbs\_left) {

trbs\_left = DWC3\_TRB\_NUM;

if (usb\_endpoint\_xfer\_isoc(dep->endpoint.desc)) { dep->busy\_slot = 1; dep->free\_slot = 1;}

else { dep->busy\_slot = 0; dep->free\_slot = 0; }

}

list\_for\_each\_entry\_safe(req, n, &**dep->request\_list**, list) {

last\_one = false;

dma\_addr\_t dma = **req->request.dma**;

unsigned length = **req->request.length;**

trbs\_left--; if (!trbs\_left) last\_one = 1;

**if (list\_is\_last(&req->list, &dep->request\_list)) last\_one = 1;**

**dwc3\_prepare\_one\_trb**(dep, req, **dma, length, last\_one**, false, 0);

if (last\_one) break;

}

}

### dwc3\_prepare\_one\_trb() //初始化struct dwc3\_trb

#### list\_move\_tail(&req->list, &dep->req\_queued)

static void dwc3\_prepare\_one\_trb(struct dwc3\_ep \*dep, struct dwc3\_request \*req, dma\_addr\_t dma,

unsigned length, unsigned last, unsigned chain, unsigned node)

{

**struct dwc3\_trb** \*trb = &dep->trb\_pool[dep->free\_slot & DWC3\_TRB\_MASK];

if (!req->trb) {

**dwc3\_gadget\_move\_request\_queued(**req);

req->trb = trb;

req->trb\_dma = dwc3\_trb\_dma\_offset(dep, trb);

req->start\_slot = dep->free\_slot & DWC3\_TRB\_MASK;

}

dep->free\_slot++; //free\_slot=[1, 100]

if (((dep->free\_slot & DWC3\_TRB\_MASK) == DWC3\_TRB\_NUM - 1) && usb\_endpoint\_xfer\_isoc(dep->endpoint.desc))

dep->free\_slot++;

**trb->size = DWC3\_TRB\_SIZE\_LENGTH(length);**

**trb->bpl = lower\_32\_bits(dma);**

**trb->bph = upper\_32\_bits(dma);**

trb->ctrl |= DWC3\_TRB\_CTRL\_HWO;

}

static inline void **dwc3\_gadget\_move\_request\_queued**(struct dwc3\_request \*req)

{

**req->queued = true**;

**list\_move\_tail(&req->list, &dep->req\_queued);**

}

## USB Data dequeue

### dwc3\_endpoint\_interrupt()

static void dwc3\_endpoint\_interrupt(struct dwc3 \*dwc, const struct dwc3\_event\_depevt \*event)

{

struct dwc3\_ep \*dep = dwc->eps[epnum];

switch (event->endpoint\_event) {

case **DWC3\_DEPEVT\_XFERINPROGRESS**:

**dwc3\_endpoint\_transfer\_complete**(dwc, dep, event, 0);

break;

}

### dwc3\_endpoint\_transfer\_complete()

void dwc3\_endpoint\_transfer\_complete(struct dwc3 \*dwc,struct dwc3\_ep \*dep, const struct dwc3\_event\_depevt \*event,int start\_new)

{

clean\_busy = **dwc3\_cleanup\_done\_reqs**(dwc, dep, event, status);

if (clean\_busy)

dep->**flags &= ~DWC3\_EP\_BUSY**;

}

### dwc3\_cleanup\_done\_reqs()

static int dwc3\_cleanup\_done\_reqs(struct dwc3 \*dwc, struct dwc3\_ep \*dep, const struct dwc3\_event\_depevt \*event, int status)

{

do {

**struct dwc3\_request** \*req = **next\_request**(&dep->req\_queued);

i = 0;

do {

slot = req->start\_slot + i;

if ((slot == DWC3\_TRB\_NUM - 1) && usb\_endpoint\_xfer\_isoc(dep->endpoint.desc)) slot++;

slot %= DWC3\_TRB\_NUM;

**struct dwc3\_trb** \*trb = &**dep->trb\_pool**[slot];

ret = **\_\_dwc3\_cleanup\_done\_trbs**(dwc, dep, req, trb, event, status);

if (ret) break;

}while (++i < req->request.num\_mapped\_sgs);

**dwc3\_gadget\_giveback**(dep, req, status);

if (ret) break;

} while (1);

if (usb\_endpoint\_xfer\_isoc(dep->endpoint.desc) && **list\_empty(&dep->req\_queued)**) {

if (**list\_empty(&dep->request\_list**)) {

dep->**flags = DWC3\_EP\_PENDING\_REQUEST**;

} else {

dwc3\_**stop\_active**\_transfer(dwc, dep->number);

dep->**flags = DWC3\_EP\_ENABLED;**

}

return 1;

}

return 1;

}

### \_\_dwc3\_cleanup\_done\_trbs()

static int \_\_dwc3\_cleanup\_done\_trbs(struct dwc3 \*dwc, struct dwc3\_ep \*dep,

struct dwc3\_request \*req, struct dwc3\_trb \*trb, const struct dwc3\_event\_depevt \*event, int status)

{

**count = trb->size & DWC3\_TRB\_SIZE\_MASK;**

if (dep->direction) {

if (count) {

trb\_status = DWC3\_TRB\_SIZE\_TRBSTS(trb->size);

if (trb\_status == DWC3\_TRBSTS\_MISSED\_ISOC) { dep->**flags |= DWC3\_EP\_MISSED\_ISOC**; }

else { **status = -ECONNRESET;**  }

} else { dep->flags &= ~DWC3\_EP\_MISSED\_ISOC; }

} else {

if (count && (event->status & DEPEVT\_STATUS\_SHORT)) s\_pkt = 1;

}

**req->request.actual += req->request.length - count**;

if (s\_pkt) return 1;

if ((event->status & DEPEVT\_STATUS\_LST) && (trb->ctrl & (DWC3\_TRB\_CTRL\_LST | DWC3\_TRB\_CTRL\_HWO))) return 1;

if ((event->status & DEPEVT\_STATUS\_IOC) && (trb->ctrl & DWC3\_TRB\_CTRL\_IOC)) return 1;

**return 0;**

}

### dwc3\_gadget\_giveback()

#### list\_del(&req->list);

void dwc3\_gadget\_giveback(struct dwc3\_ep \*dep, struct dwc3\_request \*req, int status)

{

if (**req->queued**) {

i = 0;

do {

**dep->busy\_slot++**;

if (((dep->busy\_slot & DWC3\_TRB\_MASK) == DWC3\_TRB\_NUM- 1) && usb\_endpoint\_xfer\_isoc(dep->endpoint.desc))

dep->busy\_slot++;

} while(++i < req->request.num\_mapped\_sgs); // i=0

**req->queued = false;**

}

**list\_del(&req->list);**

req->trb = NULL;

if (req->request.status == -EINPROGRESS) req->request.status = status;

if (dwc->ep0\_bounced && dep->number == 0) dwc->ep0\_bounced = false;

else **usb\_gadget\_unmap\_request**(&dwc->gadget, &req->request, req->direction);

**req->request.complete**(&dep->endpoint, &req->request);

}

# Buffer Management

## Buffer流程

### vb2\_buffer

用户填充vb2\_buffer数据后：

1. 设置v4l2\_planes[0].bytesused = b->bytesused，vb2\_buffer的实际大小
2. 将vb2\_buffer. queued \_entry挂到vb2\_queue. queued \_list中
3. 设置vb2\_buffer状态为VB2\_BUF\_STATE\_QUEUED
4. 设置uvc\_buffer的状态为UVC\_BUF\_STATE\_QUEUED
5. 设置uvc\_buffer->mem，uvc\_buffer->length，uvc\_buffer->byteused

vb2\_buffer中的数据处理中:

1. 设置vb2\_buffer状态为VB2\_BUF\_STATE\_ACTIVE
2. 将uvc\_buffer.queue挂到uvc\_video\_queue.irqqueue中

vb2\_buffer中的数据处理后:

1. 将uvc\_buffer.queue从uvc\_video\_queue.irqqueue中移除
2. 设置vb2\_buffer状态为VB2\_BUF\_STATE\_DONE
3. 将vb2\_buffer.done\_entry挂到vb2\_queue.done\_list中
4. 触发vb2\_queue.done\_wq中断

用户获取vb2\_buffer:

1. vb2\_buffer.done\_entry从vb2\_queue.done\_list中移除
2. vb2\_buffer.queued\_entry从vb2\_queue.queued\_list中移除
3. 设置vb2\_buffer状态为VB2\_BUF\_STATE\_DEQUEUED

### uvc\_buffer

获取uvc\_buffer的数据

1. 从uvc\_video\_queue.irqqueue中获取uvc\_buffe.queue
2. usb\_request.list从uvc\_video.req\_free中移除
3. 将uvc\_buffe中的数据拷贝到usb\_request中

## Video init

### struct uvc\_buffer buf

**struct uvc\_buffer** {

**struct vb2\_buffer buf;**

**struct list\_head queue;** //list\_add\_tail(&buf->queue, &queue->irqqueue)

enum uvc\_buffer\_state **state**; //UVC\_BUF\_STATE\_QUEUED

void \***mem**; //vb->planes[plane\_no].mem\_priv

unsigned int **length**; // vb->v4l2\_planes[plane\_no].length

unsigned int **bytesused**; //vb->v4l2\_planes[plane\_no].bytesused;

};

### struct uvc\_video video

**struct uvc\_video** {

**struct usb\_ep \*ep;**

u8 bpp;

u32 fcc;

unsigned int **width**;

unsigned int **height**;

unsigned int **imagesize**;

unsigned int **req\_size**;

// req\_size = video->ep->maxpacket \* max\_t(unsigned int, video->ep->maxburst, 1) \* (video->ep->mult + 1)

**struct usb\_request \*req[UVC\_NUM\_REQUESTS];** //UVC\_NUM\_REQUESTS = 4

\_\_u8 **\*req\_buffer[UVC\_NUM\_REQUESTS]**; //kmalloc(req\_size)

**struct list\_head req\_free;**  //list\_add\_tail(&video->req[i]->list, &video->req\_free);

spinlock\_t req\_lock;

void (\*encode)(); //uvc\_video\_encode\_isoc()

\_\_u32 payload\_size;

\_\_u32 max\_payload\_size;

**struct uvc\_video\_queue queue;**

unsigned int fid;

};

### struct uvc\_video\_queue queue

**struct uvc\_video\_queue** {

**struct vb2\_queue queue;**

struct mutex mutex;

unsigned int flags;

\_\_u32 **sequence**; //0

unsigned int **buf\_used**; //0

spinlock\_t irqlock;

**struct list\_head irqqueue;**  //list\_add\_tail(&buf->queue, &queue->irqqueue)

};

### struct vb2\_queue q

**struct vb2\_queue** {

enum v4l2\_buf\_type type;

unsigned int io\_modes;

unsigned int io\_flags;

struct mutex \*lock;

struct v4l2\_fh \*owner;

const struct vb2\_ops \*ops;

const struct vb2\_mem\_ops \*mem\_ops;

void \*drv\_priv;

unsigned int buf\_struct\_size;

u32 timestamp\_type;

gfp\_t gfp\_flags;

enum v4l2\_memory memory;

**struct vb2\_buffer \*bufs[VIDEO\_MAX\_FRAME];** //vb[req->count]

unsigned int **num\_buffers;** //req->count

**struct list\_head queued\_list;** //list\_add\_tail(&vb->queued\_entry, &q->queued\_list);

atomic\_t queued\_count;

**struct list\_head done\_list;**

spinlock\_t done\_lock;

wait\_queue\_head\_t done\_wq;

void \*alloc\_ctx[VIDEO\_MAX\_PLANES];

unsigned int **plane\_sizes[VIDEO\_MAX\_PLANES];** //sizes[0] = video->imagesize

unsigned int **streaming**:1; //1

struct vb2\_fileio\_data \*fileio;

};

### struct vb2\_buffer vb

**struct vb2\_buffer** {

**struct v4l2\_buffer v4l2\_buf;**

**struct v4l2\_plane v4l2\_planes[VIDEO\_MAX\_PLANES];**

**struct vb2\_queue \*vb2\_queue;** //q

unsigned int num\_planes; //1

enum vb2\_buffer\_state state; //VB2\_BUF\_STATE\_DEQUEUED 🡪 VB2\_BUF\_STATE\_PREPARED 🡪

VB2\_BUF\_STATE\_ACTIVE 🡨 VB2\_BUF\_STATE\_QUEUED 🡨

**struct list\_head queued\_entry;** //list\_add\_tail(&vb->queued\_entry, &q->queued\_list);

**struct list\_head done\_entry;**

**struct vb2\_plane planes[VIDEO\_MAX\_PLANES];** //vb->planes[plane].mem\_priv = mem\_priv

};

### struct v4l2\_buffer b

**struct v4l2\_buffer** {

\_\_u32 index;

\_\_u32 type;

\_\_u32 bytesused; //vb->v4l2\_planes[0].bytesused

\_\_u32 flags;

\_\_u32 field;

struct timeval timestamp;

struct v4l2\_timecode timecode;

\_\_u32 sequence;

\_\_u32 memory;

union {

\_\_u32 offset; //vb->v4l2\_planes[0].m.mem\_offset

unsigned long userptr;

struct v4l2\_plane \*planes;

\_\_s32 fd;

} m;

\_\_u32 **length**; //vb->v4l2\_planes[0].length

\_\_u32 reserved2;

\_\_u32 reserved;

};

### struct v4l2\_plane v4l2\_planes

**struct v4l2\_plane** {

\_\_u32 bytesused; //b->bytesused

\_\_u32 **length**; //q->plane\_sizes[plane]

union {

\_\_u32 mem\_offset;

unsigned long userptr;

\_\_s32 fd;

} m;

\_\_u32 data\_offset; //0

\_\_u32 reserved[11];

};

### struct vb2\_plane

**struct vb2\_plane** {

void \***mem\_priv**; //malloc(q->plane\_sizes[plane])

struct dma\_buf \*dbuf;

unsigned int dbuf\_mapped;

};

### struct usb\_request request

**struct usb\_request** {

void \*buf; //video->req\_buffer[i]

unsigned length; //0

dma\_addr\_t dma; // dma\_map\_single(req->buf, req->length, DMA\_FROM\_DEVICE)

struct scatterlist \*sg;

unsigned num\_sgs;

unsigned num\_mapped\_sgs;

unsigned stream\_id:16;

unsigned no\_interrupt:1;

unsigned zero:1;

unsigned short\_not\_ok:1;

void (\*complete)(); //uvc\_video\_complete()

void \*context; // struct uvc\_video video

**struct list\_head list;** //list\_add\_tail(&video->req[i]->list, &video->req\_free);

int status;

unsigned **actual**; //0

};

### struct dwc3\_request req

**struct dwc3\_request** {

**struct usb\_request request;**

**struct list\_head list; //list\_add\_tail(&req->list, &dep->request\_list);**

**//list\_move\_tail(&req->list, &dep->req\_queued)**

**struct dwc3\_ep \*dep;**

u32 start\_slot; //dep->free\_slot

u8 **epnum**; //dep->number

**struct dwc3\_trb \*trb; // dep->trb\_pool[dep->free\_slot]**

dma\_addr\_t trb\_dma;

unsigned **direction**:1; // dep->direction

unsigned mapped:1;

unsigned queued:1;

};

### struct dwc3\_ep dep

**struct dwc3\_ep** {

**struct usb\_ep endpoint;**

**struct list\_head request\_list; //** **list\_add\_tail(&req->list, &dep->request\_list);**

**struct list\_head req\_queued; //list\_move\_tail(&req->list, &dep->req\_queued)**

**struct dwc3\_trb \*trb\_pool; //trb = &dep->trb\_pool[dep->free\_slot]**

dma\_addr\_t trb\_pool\_dma; // **dep->trb\_pool的dma对应的handle**

u32 **free\_slot**; //free\_slot++

u32 **busy\_slot**;

const struct usb\_ss\_ep\_comp\_descriptor \*comp\_desc;

**struct dwc3 \*dwc;**

unsigned flags;

unsigned current\_trb;

u8 **number**;

u8 type;

u8 resource\_index;

u32 interval;

char name[20];

unsigned **direction**:1;

unsigned stream\_capable:1;

};

#define DWC3\_TRB\_NUM (32)

trb\_pool = dma\_alloc\_coherent(dwc->dev,sizeof(struct dwc3\_trb) \* DWC3\_TRB\_NUM,&dep->trb\_pool\_dma, GFP\_KERNEL);

### struct dwc3\_trb trb

trb: transfer request block

struct dwc3\_trb {

u32 bpl; //DW0-3 // lower\_32\_bits(req->request.dma)

u32 bph; //DW4-7 // upper\_32\_bits(req->request.dma)

u32 size; //DW8-B // DWC3\_TRB\_SIZE\_LENGTH(req->request.length)

u32 ctrl; //DWC-F

} \_\_packed;

## Video qbuf

### Buffer init

INPUT : b->bytesused

============================================================

QBUF:

============================================================

**v4l2\_planes[0].bytesused** = b->bytesused; //imgsize

v4l2\_planes[0].data\_offset = 0;

**list\_add\_tail(&vb->queued\_entry, &q->queued\_list);**

**vb->state** = VB2\_BUF\_STATE\_ACTIVE;

**list\_add\_tail(&buf->queue, &queue->irqqueue);**

### Buffer move

============================================================

buf->bytesused = imgsize = 320x240\*2 = 153600 : (YUYV)

queue->buf\_used = 1022\*n = 0: (n=0~149)

============================================================

**struct usb\_request \*req** = list\_first\_entry(&video->req\_free, struct usb\_request, list);

**list\_del(&req->list);**

**struct uvc\_buffer \*buf** = list\_first\_entry(&queue->irqqueue, struct uvc\_buffer, queue);

data = **req->buf**;

len = **video->req\_size**; //1024

**data[0] = 0x02;**

**data[1] = UVC\_STREAM\_EOH | video->fid;**

data += 2;

len -= 2; //1022

mem = buf->mem + queue->buf\_used;

nbytes = min((unsigned int)len, buf->bytesused - queue->buf\_used); //1022

**memcpy(data, mem, nbytes);**

**queue->buf\_used** += nbytes; // += 1022

len -= nbytes; //0

**req->length** = video->req\_size - len; //1024

**usb\_ep\_queue(video->ep, req, GFP\_ATOMIC);**

============================================================

buf->bytesused = imgsize = 320x240\*2 = 153600 : (YUYV)

queue->buf\_used = 1022\*n = 153300: (n=150)

============================================================

**struct usb\_request \*req** = list\_first\_entry(&video->req\_free, struct usb\_request, list);

**list\_del(&req->list);**

**struct uvc\_buffer \*buf** = list\_first\_entry(&queue->irqqueue, struct uvc\_buffer, queue);

data = **req->buf**;

len = video->req\_size; //1024

**data[0] = 0x02;**

**data[1] = UVC\_STREAM\_EOH | video->fid | UVC\_STREAM\_EOF;**

data += 2;

len -= 2; //1022

mem = buf->mem + queue->buf\_used;

nbytes = min((unsigned int)len, buf->bytesused - queue->buf\_used); //153600-153300=300

**memcpy(data, mem, nbytes);**

**queue->buf\_used** += nbytes; // += 300

len -= nbytes; //1022-300 = 722

**req->length** = video->req\_size - len; //1024 - 722 = 302

video->queue.buf\_used = 0;

**buf->state** = UVC\_BUF\_STATE\_DONE;

**video->fid** ^= UVC\_STREAM\_FID;

**list\_del(&buf->queue);**

**vb->state** = VB2\_BUF\_STATE\_DONE

**list\_add\_tail(&vb->done\_entry, &q->done\_list);**

**wake\_up(&q->done\_wq);**

**usb\_ep\_queue(video->ep, req, GFP\_ATOMIC);**

## Video dqbuf

wait\_event\_interruptible(**q->done\_wq,** !list\_empty(&q->done\_list) || !q->streaming)

**struct vb2\_buffer \*vb** = list\_first\_entry(&q->done\_list, struct vb2\_buffer, done\_entry);

**list\_del(&(\*vb)->done\_entry);**

**list\_del(&vb->queued\_entry);**

**vb->state** = VB2\_BUF\_STATE\_DEQUEUED;

struct dwc3\_trb {

u32 bpl;

u32 bph;

u32 size;

u32 ctrl;

} \_\_packed;

# End